下载APP

20｜滑动窗口：TCP是如何进行流量控制和拥塞控制的？

2022-01-25 黄清昊

《算法实战高手课》 课程介绍 >

讲述：黄清昊
时长 16:17 大小 14.92M

你好，我是微扰君。

过去几讲，我们一起讨论了最短路算法在网络中的应用，学习了从 Dijkstra 算法思想发展而来的链路状态选路算法，以及从 Bellman-Ford 算法思想发展而来的距离矢量算法。

链路状态算法的每个节点，通过通信，都构建了完整的网络拓扑图，然后根据 Dijkstra 算法独立地计算最短路径，并依据计算结果维护动态路由表；距离矢量算法，则是通过节点间的通信了解邻居到每个不同节点的距离，以此作为选路依据，所以链路上传输的压力比链路状态算法小了很多，但也因为没有全局的信息，网络出现故障时很容易陷入无穷计问题。

在计算机网络发展以来，类似单源最短路问题的图论算法应用，除了这两大经典算法，其实还有很多，比如最小生成树问题、网络流问题等等，它们都在不同的场景下发挥着巨大的作用，但我们要知道，图论算法也只是解决了网络传输中和“拓扑结构”相关的一小部分问题。

这些算法并不足以让我们的数据在环境复杂的网络上稳定传输，也并没有办法去控制流量传输的快慢，来避免接受方对数据处理不过来，或者网络上数据包太多产生拥塞的情况。为了解决传输本身的问题，自然也有一些经典的算法思想和协议被提出来，TCP 中的滑动窗口、拥塞窗口就是经典的例子。

在学习具体算法之前，我们先简单复习一下 TCP 协议做到了什么。

TCP 协议和 UDP 协议

TCP 作为最常用的两大传输层协议之一，无疑是久经生产环境检验的。传输层有两个我们广泛使用的协议：UDP 协议、TCP 协议，我们一般会说前者是面向无连接的，后者是面向连接的。

这里的“连接”具体是什么意思呢？

简单来说，UDP 协议是一种没有状态的协议，节点之间如果采用 UDP 协议通信，两个节点能做的就是在 UDP 协议上发送一个个数据包，协议本身不会关心这些包之间的关系，所以在复杂的网络下，包的顺序和可达性都是没有保证的，应用层需要自己处理这些包的丢失和乱序问题。

当然，UDP 协议也因此可以设计的非常轻量，所以**在网络传输本来就比较稳定的内网环境下，或者对丢包可以容忍但对时延要求较高的场景下，UDP 协议有许多应用。**

而 TCP 很不一样，它在节点之间建立了真正的连接的概念。相信你肯定听过 TCP 的三次握手吧，UDP 就没有这个过程，三次握手完成时，TCP 连接就建立了；结束通信的时候，通信双方往往也需要主动断开连接。

TCP 的传输基于字节流，也引入了状态，明确记录着每个包是否发送、是否被接受到、包本身的序列号等状态；所以，**TCP 为我们提供了可靠有序的传输能力，也被设计的相当复**

杂。TCP 协议不止考虑了包的可靠传输，同时也兼顾了效率，更提供了对流量控制和拥塞避免的能力，滑动窗口和拥塞窗口就是为了这两个目的而设计的。

那 TCP 的具体是如何做到让网络中的包传输可靠且效率高的呢？

TCP 中包的发送

一个包在网络中发送出去，其实就像古时候你给别人用信鸽寄了一封信一样，在外界环境非常复杂的情况下，你完全没有把握这封信能不能真的送达收信人那边，除非有一天你收到了收信人的回信。

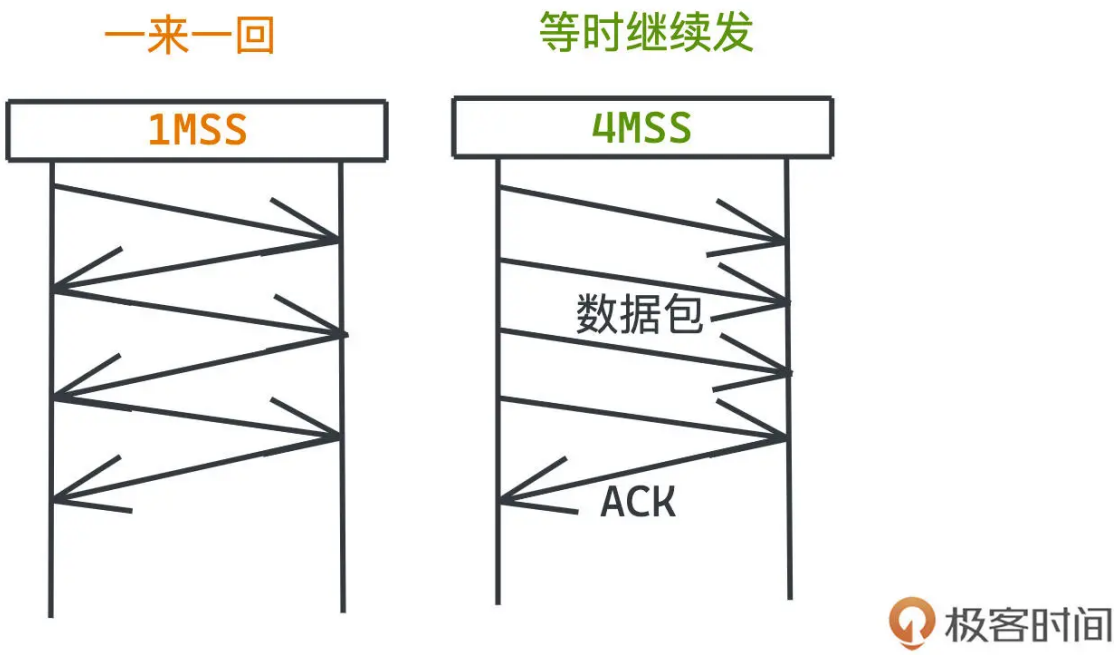
类似的，在复杂网络环境下，TCP 为了能保证每个包真的送达了，并且接收端收到包的顺序和发送端是一致的，我们每发出一个包，自然也需要一个类似回信的机制。

极客时间

这个回信也就是 ACK 包，**每个包发送的时候会有一个序列号，接收端回 ACK 包的时候会把序列号 +1 发送回来**，发送端如果没有收到某个包的 ACK 包，会在一段时间之后尝试重新发送，直到收到 ACK 为止。这其实也是在网络和各种分布式系统中能确保消息可达的唯一方式。

那问题来了，我们为了确保消息保序可达，难道每次发送一个新的包，都等待上一个包的 ACK 回来之后才能发送吗？这样一来一回的效率显然是很低的，也就是每经过一个 RTT 的时间，我们只能发送一个包，假设一个 RTT 是 100ms，那在一秒中我们甚至只能发送 10 个包，这完全是不可接受的。

其实我们**在等待 ACK 的时候没有必要停止后续包的发送，因为网络传输虽然不稳定，但大部分包往往还是可达的**，这样我们就可以获得数倍的传输效率提升。如果真的不幸遇到了丢包，接收端 ACK 姗姗来迟的时候，也就告诉了我们某个序列号之前的所有包全部收到，我们再根据一定的策略，尝试重新发送对应丢失的包就可以了。



所以自然而然的，发送方需要缓存已发出但尚未收到 ACK 的包，接收方收到包但没有被用户进程消费之前也得把收到的包留着。

但是，缓存是有大小限制的，程序消费数据和链路传输数据的能力也是有限的，发送端和接受端都需要一种机制来限制可发送或者可接收数据的最大范围。

于是，滑动窗口和拥塞窗口应运而生。

这两个算法核心都是为了防止像网络中发送的包太多。不同的是两者的目的，**滑动窗口机制，可以用来控制流量，防止接收方处理不过来消息；同样基于窗口机制的拥塞控制算法，则用来处理网络上数据包太多的情况，以避免网络中出现拥塞。**

流量控制

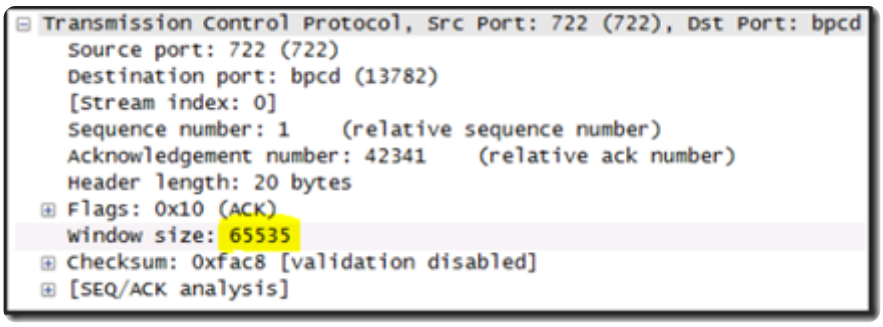
我们先来看看如何用滑动窗口控制流量。

这里说流量控制，主要就是为了防止接收方处理数据的速度跟不上发送方，避免随着时间推移，数据自然溢出接收方的缓冲区。

虽然协议可以保证发送方没有收到 ACK，最终会重试重新发送，但如果需要大量反复发送冗余的数据，所占用的网络资源就被白白浪费了，在网络资源很紧缺的时候，这也会造成网络环境的恶化。

TCP 控制流量的方式也很简单，就是滑动窗口机制。

接收端会建立一个滑动窗口，由接收方向发送方通告，TCP 首部里的 window 字段就是用来表示窗口大小的，窗口表示的就是接收方目前能接收的缓冲区的剩余大小。

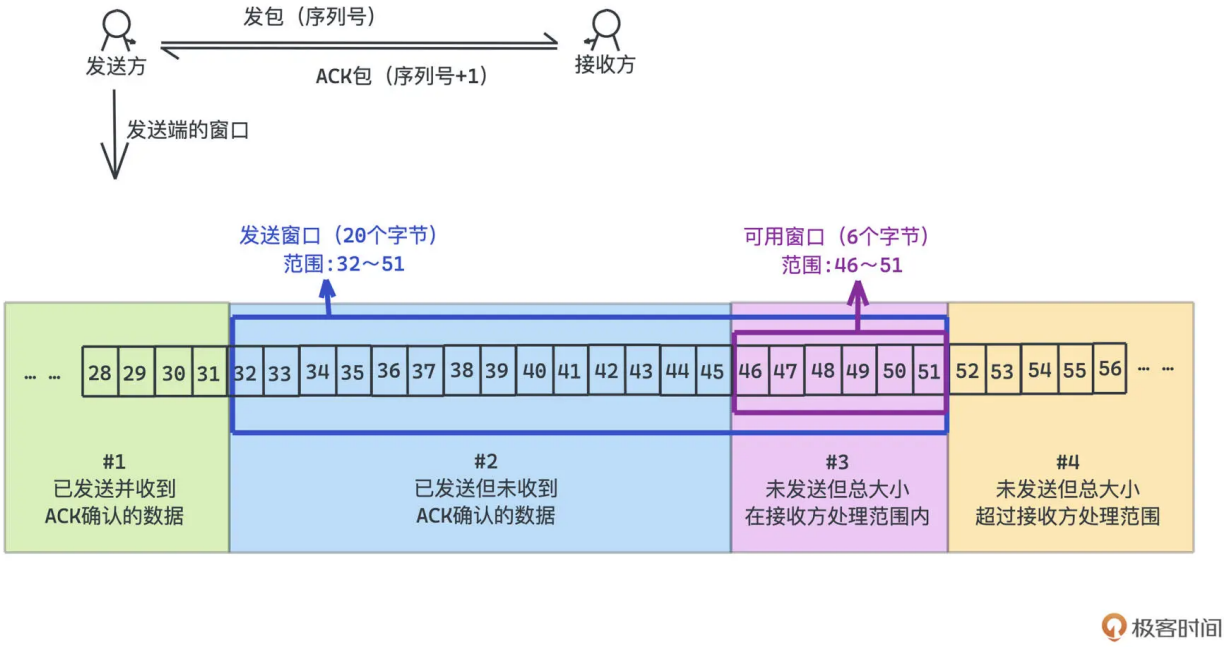


但是发送方也会根据这个通告窗口的大小建立自己的滑动窗口。为了兼顾效率和可靠性，在发送方，所有未收到 ACK 的消息虽然可以发送，但是在收到 ACK 之前是一定要在缓冲区中保存的。

我们一起来看一下。

发送端的窗口

发送窗口根据三个标准来划分：是否发送、是否收到 ACK、是否在接收方通告处理范围内，分成了四个部分。



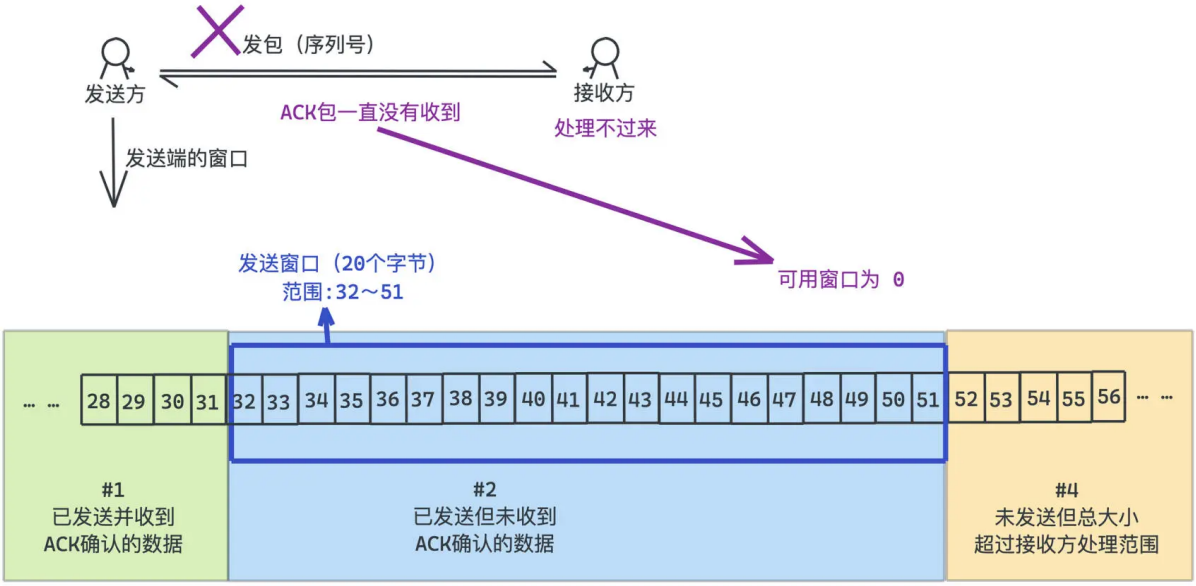
第一部分就是已经发送且收到 ACK 的部分，这一块我们知道已经成功发送，所以不需要在缓冲区保留了。

第二部分是已发送但尚未收到 ACK 的部分。

第三部分是还没有发送，但是还在接收方通告窗口也就是处理范围内的数据，这块我们也可以称为可用窗口；第二、第三部分一起构成了我们的整个发送窗口。

最后一部分则是我们**需要发送，但已经超过接收方通告窗口范围的部分**，这一部分在没有收到新的 ACK 之前，发送方是不会发送这些数据的。通过这个限制，发送的数据就一定不会超过接收方的缓冲区了。

但如果发送方一直没有收到 ACK，随着数据不断被发送，很快可用窗口就会被耗尽。在这种情况下，发送方也就不会继续发送数据了，这种发送端可用窗口为零的情况我们也称为“零窗口”。



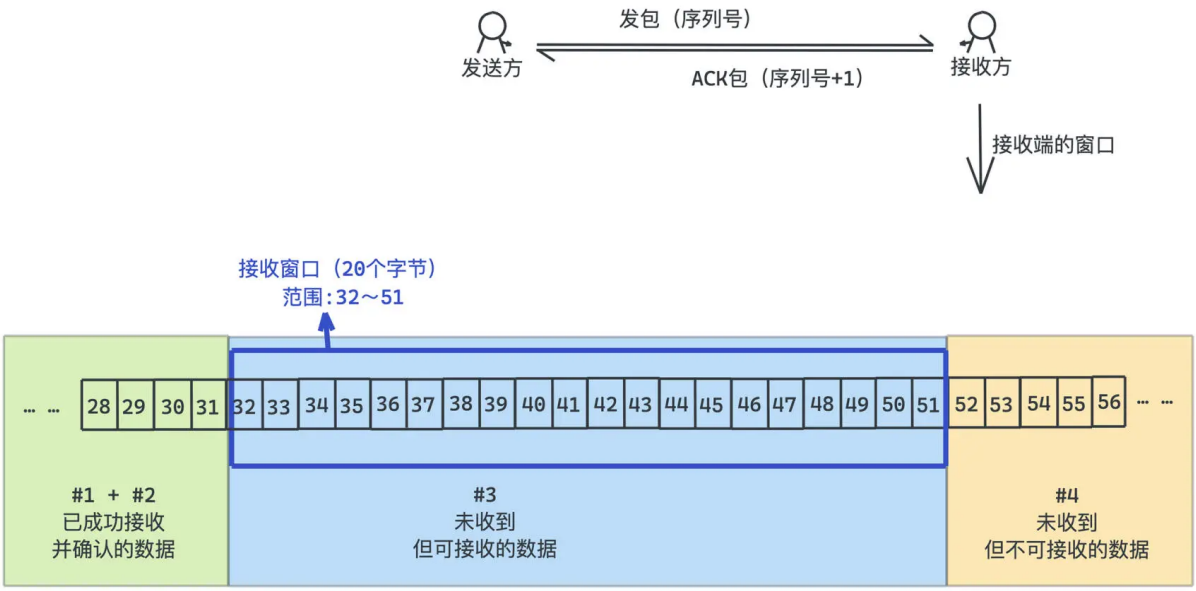
正常来说，等接收端处理了一部分数据，又有了新的可用窗口之后，就会再次发送 ACK 报文通告发送端自己有新的可用窗口（因为发送端的可用窗口是受接收端控制的）。

但是，万一要是 ACK 消息在网络传输中正好丢包了，那发送端还能感知到接收端窗口的变化吗？其实是不会的，在这个情况下，接收端就会一直等着发送端发送数据，而发送端也还会以为接收端仍然处于零窗口的状态，这样一直互相等待，就好像进入了死锁状态。

解决办法也很简单，我们可以再引入一个零窗口定时器，如果发送端陷入零窗口的状态，就会启动这个定时器，去定时地询问接收端窗口是否可用了，这也是在分布式系统中常见的处理丢包的方式之一。

接收端的窗口

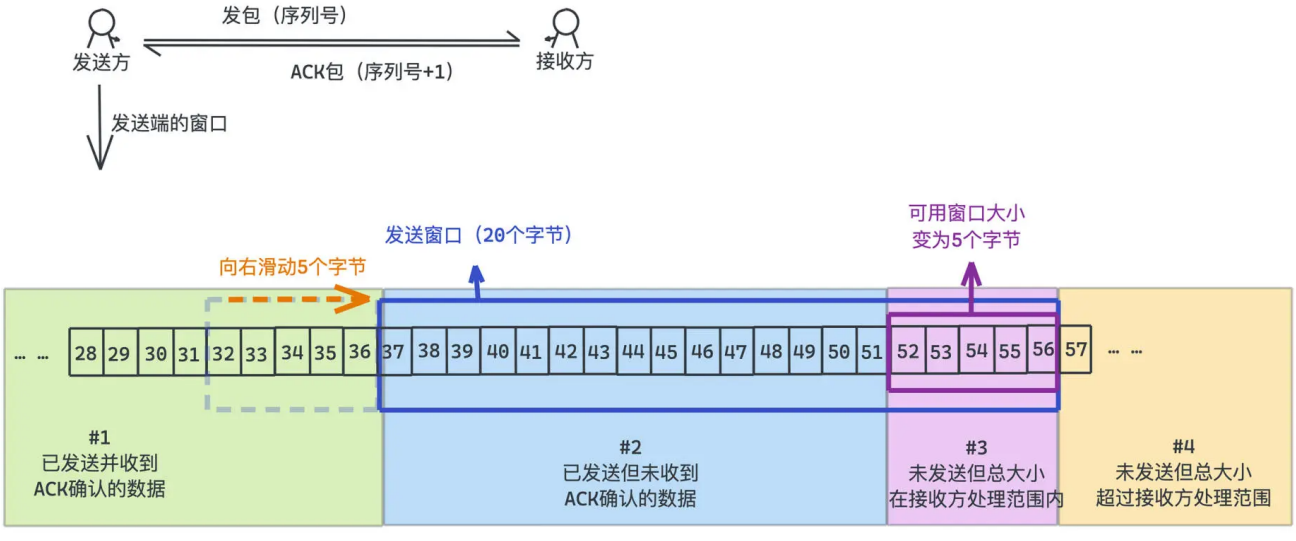
相对发送端来说，接收端要简单的多，主要就分为已经接收并确认的数据和未收到但可以接收的数据，这一部分也就是接收窗口；剩下的就是缓冲区放不下的区域，也就是不可接收的区域。



极客时间

如果进程读取缓冲区速度有所变化，接收端可能也会改变接收窗口的大小，每次通告给发送端，就可以控制发送端的发送速度了。这就是所谓的滑动窗口，也就是流量控制机制。

而之所以是滑动窗口，也很好理解，随着 ACK 或者进程读取数据，窗口也会顺次往后移动。比如在发送端的窗口中，如果我们在某次通信中收到了一条 ACK 消息，表示 36 之前的消息都已经被收到了，那么整个可用的窗口就会顺次往右移动。

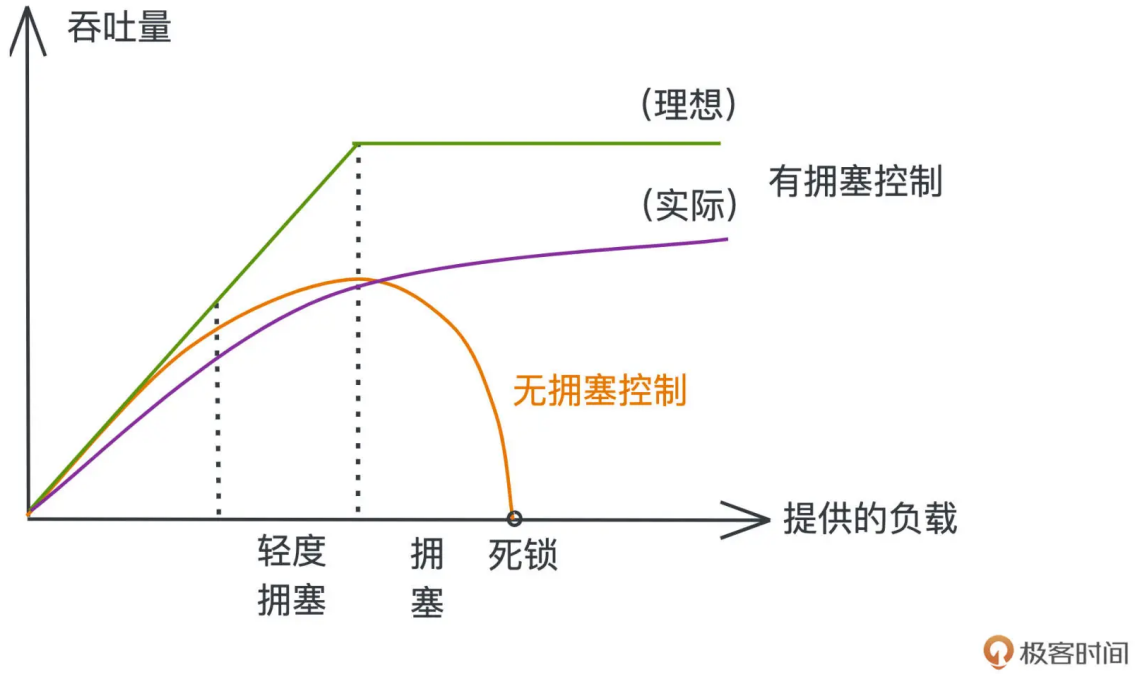


极客时间

总的来说，滑动窗口（流量控制机制）解决了发送端消息可能淹没接收端，导致处理跟不上的情况。

流量拥塞

那 TCP 协议如何解决流量拥塞的情况呢？也就是网络中由于大量包传输，导致吞吐量下降甚至为 0 的情况。这和我们的道路交通很像，当车流越来越大的时候，整体的行车速度可能会不断下降，导致拥堵，最后吞吐量反而不如车少的时候。



在实际网络中，因为大量的包传输，可能导致中间某些节点的缓冲区满载，从而多余的包被丢弃，需要重新发送，情况越发恶化，最差的时候，网络上的包都是重传的包并且反复地丢弃；整个网络传输能力甚至可以降低为 0。

这当然是一个很严重的问题，TCP 协议同样提出了另外一个叫拥塞窗口的机制，很好地解决了这个问题。具体是怎么做的呢？我们一起来看一下。

拥塞控制

网络中每个节点不会有全局的网络通信情况，唯一能发现的就是自己的部分包丢了，这种时候它就有理由怀疑网络环境劣化，可能产生了拥塞。

TCP 是一个比较无私的协议，在这种情况下，会选择减少自己发送的包。当网络上大部分通信协议传输层都采用的是 TCP 协议时，在出现拥塞的情况下，大部分节点都会不约而同地减少自己传输的包，这样网络拥塞情况就会得到极大的缓解，一直处于比较好的网络状态。

所以我们就需要**在发送端定义一个窗口 CWND（congestion window），也就是拥塞窗口；发送端能发送的最多没有收到 ACK 的包，也不会超过拥塞窗口的范围。**

引入拥塞控制机制的 TCP 协议，发送端最大的发送范围是拥塞窗口和滑动窗口中小的一个。拥塞窗口会动态地随着网络情况的变化而进行调整，大体上的策略是如果没有出现拥塞，我们扩大窗口大小，否则就减少窗口大小。

具体是如何实现的呢？经典拥塞控制算法主要包括四个部分：

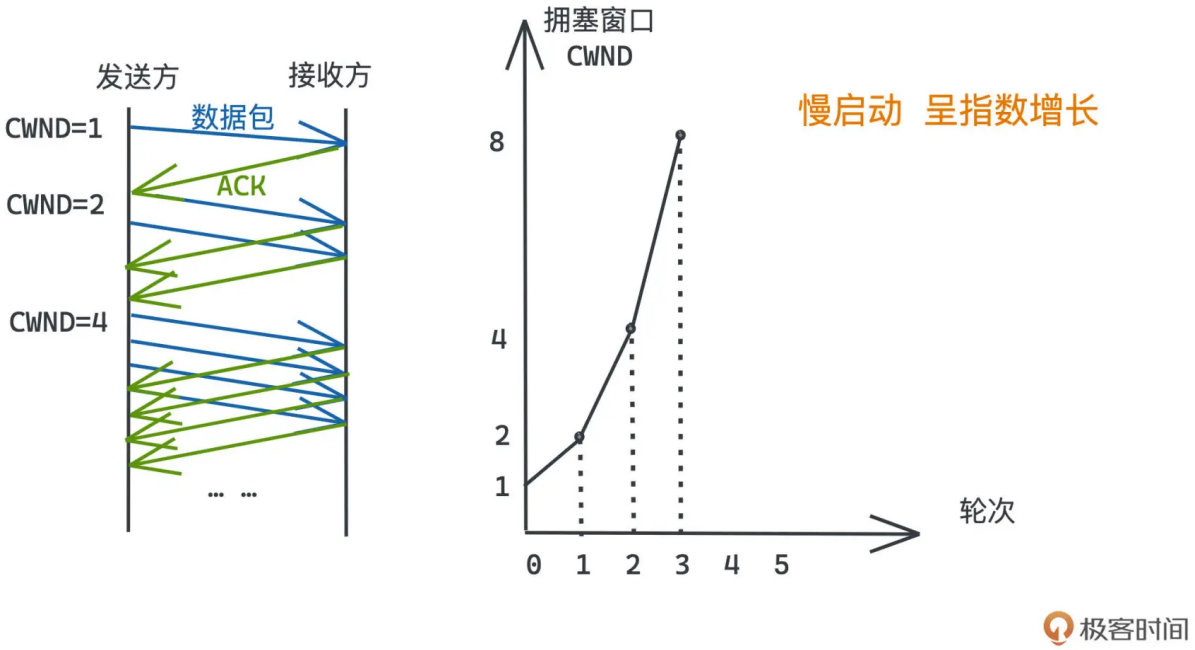
慢启动

拥塞避免

拥塞发生

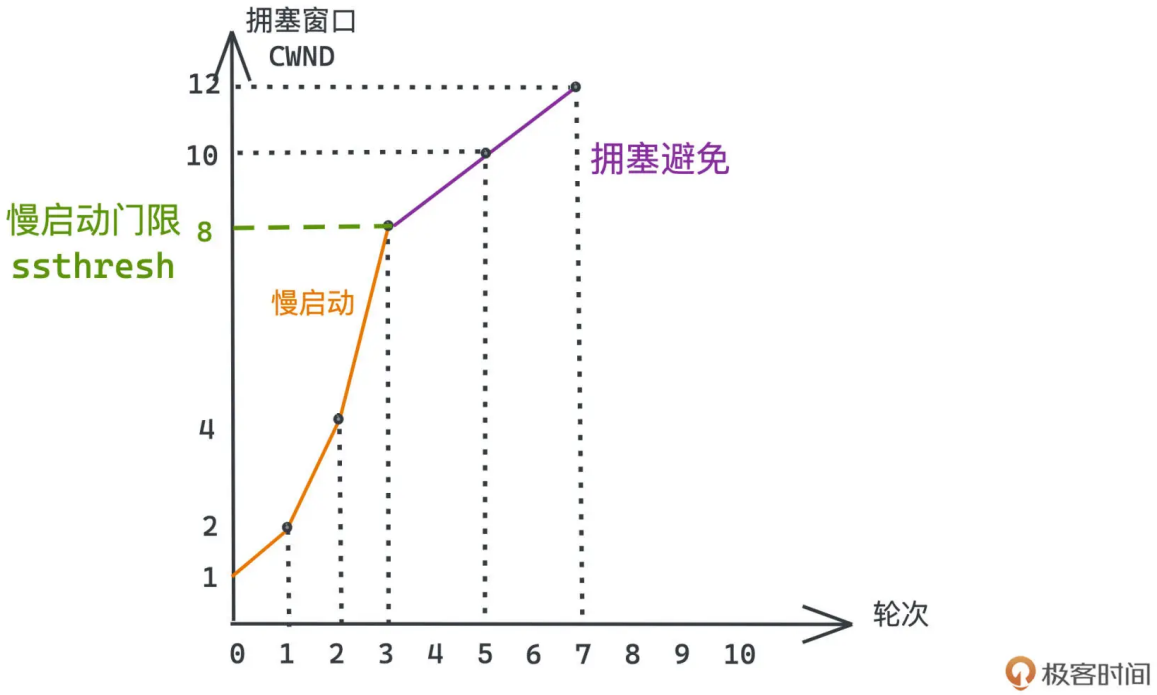
快速恢复

我们一个个来看。首先是慢启动，在不确定拥塞是否会发生的时候，我们不会一上来就发送大量的包，而是会采用倍增的方式缓慢增加窗口的大小，窗口大小从 1 开始尝试，然后尝试 2、4、8、16 等越来越大的窗口。



整个慢启动的过程看起来就像图中这样，指数型的增加拥塞窗口的大小。

这样，倍增的方式窗口就会很快扩大；我们会在窗口大到一定程度时，减慢增加的速度，转成线性扩大窗口的方式，也就是每次收到新的 ACK 没有丢包的话只比上次窗口增大 1。整个过程看起来就像这样：

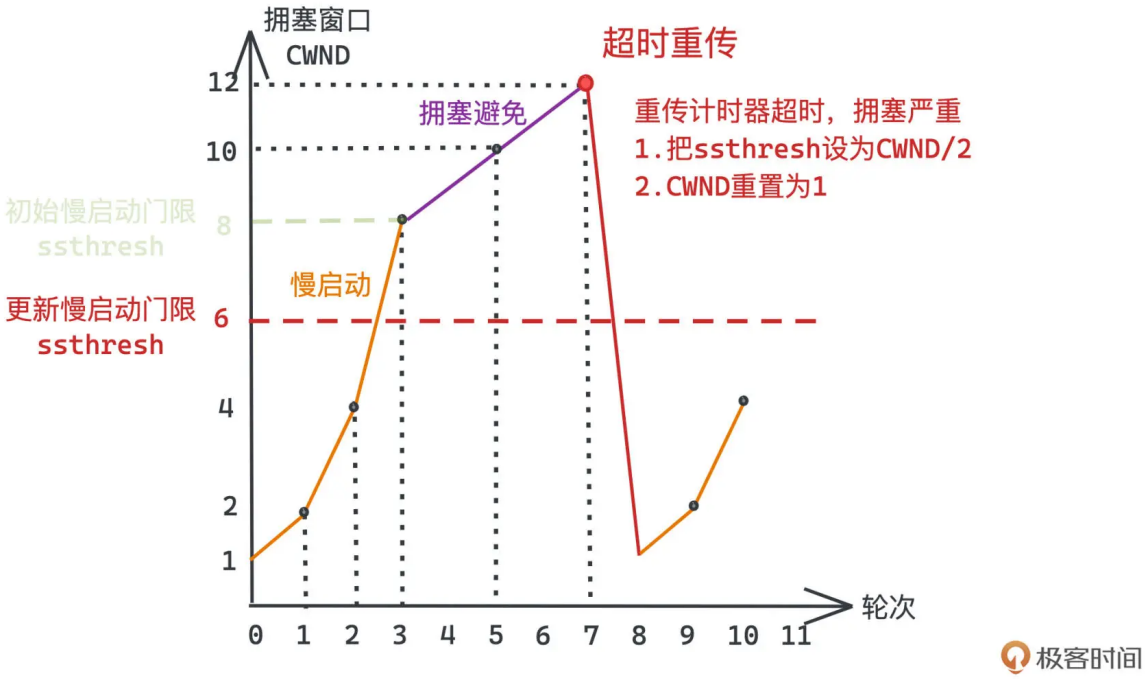


这两个慢启动阶段和拥塞避免阶段的分界点，我们就叫“慢启动门限（ssthresh）”。

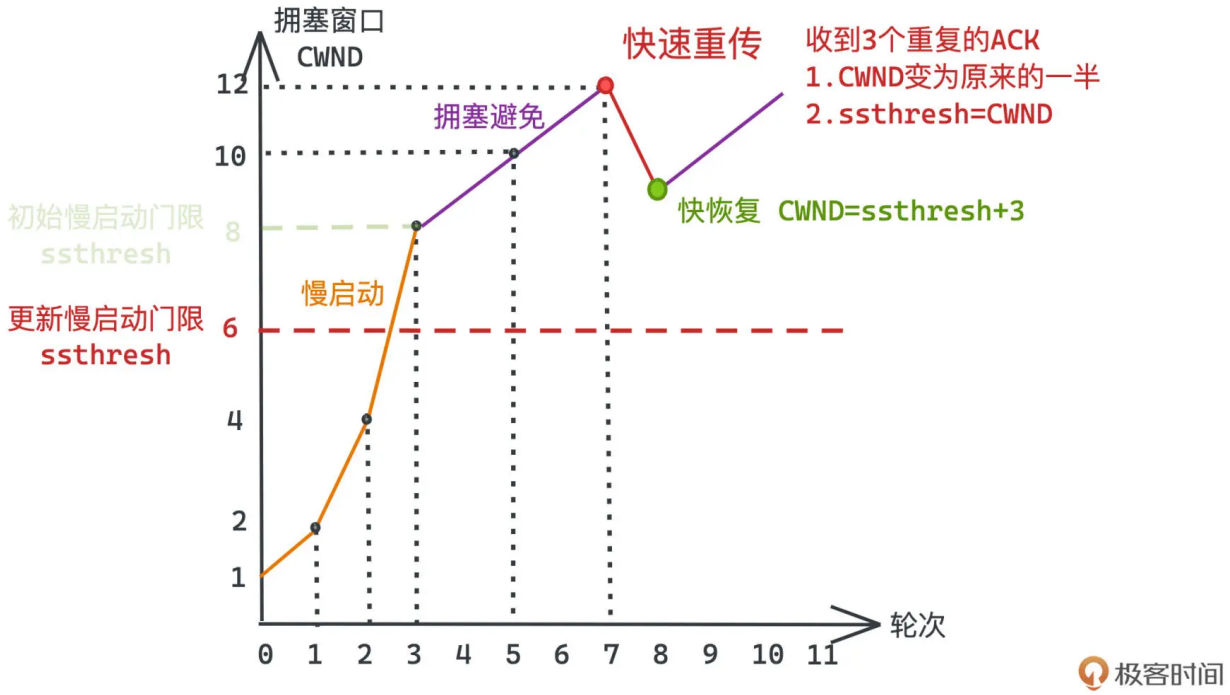
随着窗口进一步缓慢增加，终于有一天，网络还是遇到了丢包的情况，我们就会假定这是拥塞造成的。

这个时候我们一方面会进行超时重传或者快速重传，另一方面也会把窗口调整到更小的范围。

超时重传，往往意味着拥塞情况更严重，我们的策略也会更激进一些，会直接将 ssthresh 设置为重传发生时窗口大小的一半，而窗口大小直接重置为 0，再进入慢启动阶段。像这样：



快速重传，如果我们连续 3 次收到同样序号的 ACK，包还能回传，说明这个时候可能只是碰到了部分丢包，网络阻塞还没有很严重，我们就会采用柔和一点的策略，也就是快速恢复策略。



我们会先把拥塞窗口变成原来的一半，ssthresh 也就设置成当前的窗口大小，然后开始执行拥塞避免算法。有些实现也会把拥塞窗口直接设置为 ssthresh+3，本质上区别不大。

总结

总体而言，TCP 就是通过滑动窗口、拥塞窗口这两个简单的窗口实现了流量控制和拥塞控制。

滑动窗口由接收端控制，向发送端通告，这样就可以保证发送端发出的包数量上限是明确的，也就不会存在淹没接收端导致来不及处理的情况。

拥塞窗口由发送端控制，它会根据网络中的情况动态的调整，通过慢启动、拥塞避免、拥塞发生、快速恢复四个算法，就可以很好地调整窗口的大小。和滑动窗口一起限制了发送端最大的发送范围，从而保证了拥塞在网络上不会发生。

思考题

最后也给你留一个思考题。前面提到了一个叫快速重传的机制，也就是连续 3 次收到相同的 ACK，发送端就会意识到丢包的产生，我们没有详细讨论，你能说说看为什么这样比直接超时重传的方式更好吗？它有没有什么其他问题呢，会不会有更好的方式解决呢？

欢迎你留言与我一起讨论，如果觉得这篇文章对你有帮助的话，也欢迎转发给你的朋友一起学习。我们下节课见～

分享给需要的人，Ta购买本课程，你将得 20 元

生成海报并分享

赞 1 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 19 | 选路算法：距离矢量算法为什么会产生无穷计算问题？

下一篇 特别策划 | 面试：BAT面试三关准备方法大揭秘

学习推荐

精选留言

写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。